下载APP

13 | 如何给简单的图案添加纹理和复杂滤镜？

2020-07-22 月影

跟月影学可视化 进入课程 >

讲述：月影
时长 13:40 大小 12.52M

你好，我是月影。

上一课我们讲了两类处理像素的滤镜，分别是颜色滤镜和高斯滤镜。其中，**颜色滤镜是基本的简单滤镜**。因为简单滤镜里的每个像素都是独立的，所以它的处理结果**不依赖于其他像素点的信息**，因此应用起来也比较简单。**而高斯滤镜也就是平滑效果滤镜**，它是最基本的**复杂滤镜**。复杂滤镜的处理结果不仅与当前像素有关，还与其周围的像素点有关，所以应用起来很复杂。

当然了，颜色滤镜和高斯滤镜能够实现的视觉效果有限。如果想要实现更复杂的视觉效果，我们还需要使用更多其他的滤镜。所以这一节课，我们就来说说，怎么结合不同滤镜实现更复杂的视觉效果。

其他简单滤镜在 Canvas 中的应用

我们知道，简单滤镜的处理效果和像素点的颜色有关。其实，还有一些简单滤镜的处理效果和像素点的坐标、外部环境（比如鼠标位置、时间）有关。这些滤镜虽然也是简单滤镜，但能实现的效果可不简单。让我们来看几个有趣的例子。

第一个例子，实现图片边缘模糊的效果。

复制代码

```
import {loadImage, getImageData, traverse} from './lib/util.js';
const canvas = document.getElementById('paper');
const context = canvas.getContext('2d');
(async function () {
  const img = await loadImage('assets/girl1.jpg');
  const imageData = getImageData(img);
  traverse(imageData, ({r, g, b, a, x, y}) => {
    const d = Math.hypot((x - 0.5), (y - 0.5));
    a *= 1.0 - 2 * d;
    return [r, g, b, a];
  });
  canvas.width = imageData.width;
  canvas.height = imageData.height;
  context.putImageData(imageData, 0, 0);
}());
```

如上面代码所示，我们可以在遍历像素点的时候计算当前像素点到图片中心点的距离，然后根据距离设置透明度，这样我们就可以实现下面这样的边缘模糊效果了。

边缘模糊效果示意图

第二个，我们可以利用像素处理实现图片融合。比如说，我们可以给一张照片加上阳光照耀的效果。具体操作就是，把下面这张透明的 PNG 图片叠加到一张照片上。

纹理

这种能叠加到其他照片上的图片，通常被称为**纹理**（Texture），叠加后的效果也叫做纹理效果。纹理与图片叠加的代码和效果如下：

复制代码

```
import {loadImage, getImageData, traverse, getPixel} from './lib/util.js';
import {transformColor, brightness, saturate} from './lib/color-matrix.js';
const canvas = document.getElementById('paper');
const context = canvas.getContext('2d');
(async function () {
  const img = await loadImage('assets/girl1.jpg');
  const sunlight = await loadImage('assets/sunlight.png');
  const imageData = getImageData(img);
  const texture = getImageData(sunlight);
  traverse(imageData, ({r, g, b, a, index}) => {
    const texColor = getPixel(texture, index);
    return transformColor([r, g, b, a], brightness(1 + 0.7 * texColor[3]), sat
  });
  canvas.width = imageData.width;
  canvas.height = imageData.height;
  context.putImageData(imageData, 0, 0);
}());
```

阳光照耀效果图

另外，我们还可以选择不同的图片，来实现不同的纹理叠加效果，比如爆炸效果、水波效果等等。

爆炸效果图

纹理叠加能实现的效果非常多，所以它也是像素处理中的基础操作。不过，不管我们是用 Canvas 的 ImageData API 处理像素、应用滤镜还是纹理合成都有一个弊端，那就是我们必须循环遍历图片上的每个像素点。如果这个图片很大，比如它是 2000px 宽、2000px 高，我们就需要遍历 400 万像素！这个计算量是相当大的。

因为在前面的例子中，我们生成的都只是静态的图片效果，所以这个计算量的问题还不明显。一旦我们想要利用像素处理，制作出更酷炫的动态效果，这样的计算量注定会成为性能瓶颈。这该怎么办呢？

好在，我们还有 WebGL 这个神器。WebGL 通过运行着色器代码来完成图形的绘制和输出。其中，片元着色器负责处理像素点的颜色。那接下来，我们来说说如何用片元着色器处理像素。

片元着色器是怎么处理像素的？

如果想要在片元着色器中处理像素，我们需要先将图片的数据信息读取出来，交给 WebGL 程序来处理，这样我们就可以在着色器中处理了。

那么如何将图片数据信息读取出来呢？在 WebGL 中，我们会使用特殊的一种对象，叫做**纹理对象**（Texture）。我们将纹理对象作为一种特殊格式的变量，通过 uniform 传递给着色器，这样就可以在着色器中处理了。

纹理对象包括了整张图片的所有像素点的颜色信息，在着色器中，我们可以通过纹理坐标来读取对应的具体坐标处像素的颜色信息。纹理坐标是一个变量，类型是二维向量，x、y 的值从 0 到 1。在我们前面的课程里已经见过这个变量，就是我们传给顶点着色器的 uv 属性，对应片元着色器中的 vUv 变量。

因此，着色器中是可以加载纹理对象的。具体来说就是，我们先通过图片或者 Canvas 对象来创建纹理对象，然后通过 uniform 变量把它传入着色器。这样，我们再通过纹理坐标 vUv 就可以从加载的纹理对象上获取颜色信息。

1. 加载纹理

下面，我就详细说说每一步的具体操作。

首先是创建纹理对象。这个步骤比较复杂，因为设置不同的参数可以改变我们在 Shader 中对纹理取色的行为，所以其中最复杂的是参数部分。但在这里我们不需要知道太多，你先记住我在代码里给出的这几个就够了。其他的，如果之后需要用到，你再去参考相关的资料就可以了。代码如下所示：

复制代码

```
function createTexture(gl, img) {
  // 创建纹理对象
  const texture = gl.createTexture();

  // 设置预处理函数，由于图片坐标系和WebGL坐标的Y轴是反的，这个设置可以将图片Y坐标翻转一下
  gl.pixelStorei(gl.UNPACK_FLIP_Y_WEBGL, true);

  // 激活指定纹理单元，WebGL有多个纹理单元，因此在Shader中可以使用多个纹理
  gl.activeTexture(gl.TEXTURE0);

  // 将纹理绑定到当前上下文
  gl.bindTexture(gl.TEXTURE_2D, texture);

  // 指定纹理图像
  gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGBA, gl.RGBA, gl.UNSIGNED_BYTE, img);

  // 设置纹理的一些参数
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_S, gl.CLAMP_TO_EDGE);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_WRAP_T, gl.CLAMP_TO_EDGE);
  gl.texParameteri(gl.TEXTURE_2D, gl.TEXTURE_MIN_FILTER, gl.LINEAR);
}
  // 解除纹理绑定
  gl.bindTexture(gl.TEXTURE_2D, null);

```

```
  return texture;
}
```

纹理创建完成之后，我们还要**设置纹理**。具体来说就是，通过 gl.activeTexture 将对象绑定到纹理单元，再把纹理单元编号通过 uniform 写入 shader 变量中。

复制代码

```
function setTexture(gl, idx) {
  // 激活纹理单元
  gl.activeTexture(gl.TEXTURE0 + idx);
  // 绑定纹理
  gl.bindTexture(gl.TEXTURE_2D, texture);
  // 获取shader中纹理变量
  const loc = gl.getUniformLocation(program, 'tMap');
  // 将对应的纹理单元写入shader变量
  gl.uniform1i(loc, idx);
  // 解除纹理绑定
  gl.bindTexture(gl.TEXTURE_2D, null);
}
```

这样设置完成之后，我们就可以在 Shader 中使用纹理对象了。使用的代码如下：

复制代码

```
uniform sampler2D tMap;

...

vec3 color = texture2D(tMap, vUv); // 从纹理中提取颜色，vUv是纹理坐标
```

总的来说，在 WebGL 中，从创建纹理、设置纹理到使用纹理的步骤非常多，使用上可以说是非常繁琐了。方便起见，这里我们可以直接使用上一节课用过的 gl-renderer 库。经过 gl-renderer 库的封装之后，我们通过 renderer.loadTexture 就可以创建并加载纹理，然后直接将纹理对象本身作为 renderer 的 uniforms 属性值即可，就不用去关注其他细节了。具体的操作代码如下：

复制代码

```
const texture = await renderer.loadTexture(imgURL);

renderer.uniforms.tMap = texture;
```

知道了原理，接下来，我们就一起来动手把图片创建为纹理，然后加载到 Shader 中去使用吧。

首先，我们读取图片纹理并加载，代码如下所示。

复制代码

```
const texture = await renderer.loadTexture('https://p1.ssl.qhimg.com/t01cca584
renderer.uniforms.tMap = texture;

renderer.setMeshData([{
  positions: [
    [-1, -1],
    [-1, 1],
    [1, 1],
    [1, -1],
  ],
  attributes: {
    uv: [
      [0, 0],
      [0, 1],
      [1, 1],
      [1, 0],
    ],
  },
  cells: [[0, 1, 2], [2, 0, 3]],
}]);

renderer.render();
```

然后，我们直接对纹理对象取色。对应的片元着色器代码如下所示：

复制代码

```
#ifdef GL_ES
precision highp float;
#endif

uniform sampler2D tMap;
varying vec2 vUv;

void main() {
    gl_FragColor = texture2D(tMap, vUv);
}

```

在片元着色器中，我们使用 texture2D 函数来获取纹理的颜色。这个函数支持两个参数，一个是纹理单元的 uniform 变量，另一个是要获取像素的坐标，这个坐标就是我们之前用过的 uv 纹理坐标。在这个片元着色器代码里，我们只是根据 vUv 坐标将纹理图片上对应的颜色取出来，其他什么也没做，所以画布上最终呈现出来的还是原始图片。

2. 实现滤镜

加载完纹理之后，我们就可以在它的基础上实现滤镜了。用 Shader 实现滤镜的方法也很简单，为了方便你理解，这次我们就只实现图片灰度化。我们可以在前面加载纹理的基础上，引入颜色矩阵，修改后的片元着色器代码如下：

复制代码

```
#ifdef GL_ES
precision highp float;
#endif

uniform sampler2D tMap;
uniform mat4 colorMatrix;
varying vec2 vUv;

void main() {
    vec4 color = texture2D(tMap, vUv);
    gl_FragColor = colorMatrix * vec4(color.rgb, 1.0);
    gl_FragColor.a = color.a;
}
```

然后，你可以把这段代码和我们刚才加载纹理的代码做个比较。你会发现，刚才我们只是简单地把 color 从纹理坐标中取出，直接把它设置给 gl_FragColor。而现在，我们在设置 gl_FragColor 的时候，是先把颜色和 colorMatrix 相乘。这样其实就相当于是对颜色向量做了一个仿射变换。

对应地，我们修改一下前面的 JavaScript 代码。其中最主要的修改操作，就是通过 uniform 引入了一个 colorMatrix。修改后的代码如下：

复制代码

```
const texture = await renderer.loadTexture('https://p1.ssl.qhimg.com/t01cca584
renderer.uniforms.tMap = texture;
const r = 0.2126,
  g = 0.7152,
  b = 0.0722;
renderer.uniforms.colorMatrix = [
  r, r, r, 0,
  g, g, g, 0,
  b, b, b, 0,
  0, 0, 0, 1,
];

renderer.setMeshData([{
  positions: [
    [-1, -1],
    [-1, 1],
    [1, 1],
    [1, -1],
  ],
  attributes: {
    uv: [
      [0, 0],
      [0, 1],
      [1, 1],
      [1, 0],
    ],
  },
  cells: [[0, 1, 2], [2, 0, 3]],
}]);

renderer.render();
```

还记得吗？上一节课我们也实现了一个颜色矩阵，那它们有什么区别呢？区别主要有两个。

首先，上一节课的颜色矩阵是一个 4*5 的矩阵，但是因为 GLSL 语法在数据类型上不能直接支持 mat4（4*4）以上的矩阵，所以我们要计算 4*5 矩阵很不方便。而且在通常情况下，我们不经常处理颜色的 alpha 值，所以这里我就把 alpha 通道忽略了，只对 RGB 做矩阵变换，这样我们用 mat4 的齐次矩阵就够了。

其次，根据标准的矩阵与向量乘法的法则，应该是向量与矩阵的列相乘，所以我把这次传入的矩阵转置了一下，把按行排列的 rgba 换成按列排列，就得到了下面这个矩阵。

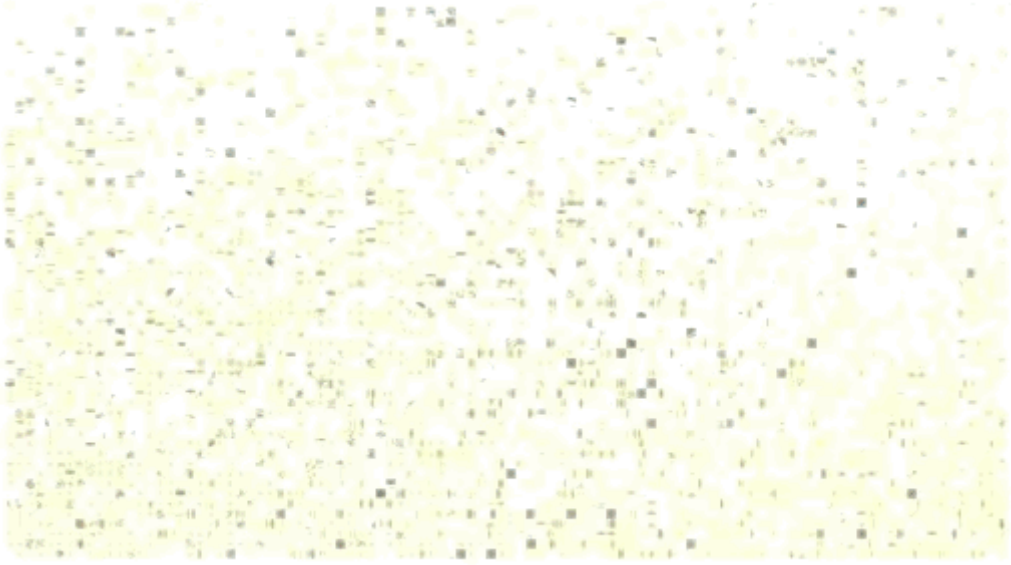
复制代码

```
renderer.uniforms.colorMatrix = [
  r, r, r, 0,
  g, g, g, 0,
  b, b, b, 0,
  0, 0, 0, 1,
];
```

这样，我们就实现了与上一节课一样的图片灰度化的功能，它是使用片元着色器实现的，在性能上要远远高于 Canvas2D。

3. 实现图片的粒子化

不过，用 Shader 只处理颜色滤镜就有些大材小用了，利用 Shader 的高性能我们可以实现一些更加复杂的效果，比如，给图片实现一个粒子化的渐显效果。如下图所示：



这个视觉效果如果在 Canvas2D 中实现，需要大量的运算，非常耗费性能，几乎不太可能流畅地运行起来，但是在 WebGL 的 Shader 中就可以轻松做到。究竟是怎么做到的呢？

我们重点来看一下 Fragment Shader 的代码。

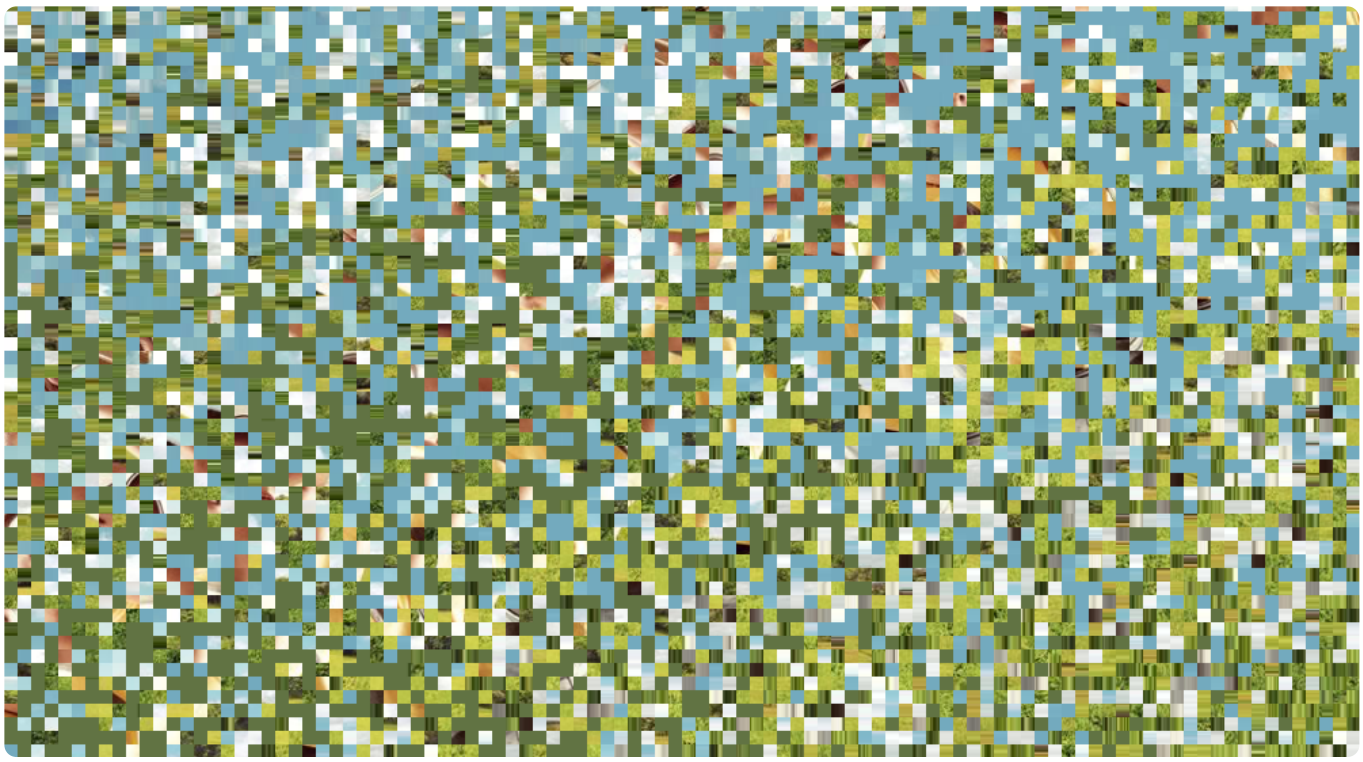
复制代码

```
#ifdef GL_ES
precision highp float;
#endif

uniform sampler2D tMap;
uniform float uTime;
varying vec2 vUv;

float random (vec2 st) {
    return fract(sin(dot(st.xy,
                         vec2(12.9898,78.233)))*
        43758.5453123);
}

void main() {
    vec2 st = vUv * vec2(100, 55.4);
    vec2 uv = vUv + 1.0 - 2.0 * random(floor(st));
    vec4 color = texture2D(tMap, mix(uv, vUv, min(uTime, 1.0)));
    gl_FragColor.rgb = color.rgb;
    gl_FragColor.a = color.a * uTime;
}
```

这段代码虽然不长，但如果你还不太熟悉 Shader，可能一眼看去，很难直接了解具体的作用，不要紧，我们一步一步来看。

首先，我们使用第 11 节课学过的重复网格技巧，将图形网格化。因为原始图像的图片像素宽高是 1000px 和 554px，所以我们用 vec2 st = vUv * vec2(100, 55.4) 就可以得到 10px X 10px 大小的网格。

然后，我们再用伪随机函数 random 根据网格随机一个偏移量，因为这个偏移量是 0~1 之间的值，我们将它乘以 2 再用 1 减去它，就能得到一个范围在 -1~1 之间的随机偏移。这样我们从纹理取色的时候，不是直接从对应的纹理坐标 vUv 处取色，而是从这个随机偏移的位置取色，就能保证取出来的颜色就是一个乱序的色值。这时候，图片显示的效果是一片随机的画面：



接着，我们引入 uTime 变量，用 mix 函数对偏移后的 uv 和原始的 vUv 相对于时间变化进行插值。当初始时间为 0 的时候，取色从 uv 取；当时间超过一个周期的时候，取色从 vUv 取；当时间在中间时，取值介于 uv 和 vUv 之间。

最后，我们再把 uTime 也和透明度关联起来。这样就实现了你上面看到的粒子化的渐显效果。

当然，这个效果做得其实还比较粗糙，因为我们引入的变量比较少，在后续的课程中，我们会一步一步深入，继续实现更加惊艳的效果。在课后，你也可以试着实现其他的效果，然后把你的成果分享出来。

4. 实现图像合成

除此之外，Fragment Shader 还可以引入多纹理，让我们可以很方便地实现图像合成。比如说，对于在电影场景合成中比较常用的绿幕图片，我们就可以使用 shader 技术把它实时地合成到其他的图像上。

举个例子，假设我们有一只猫的绿幕图片：

举个例子，现在我们有一张带有猫的绿幕图片。

带有猫的绿幕图片

我们要通过 Fragment Shader 将它合成到“高尔夫”那张照片上，具体的 shader 代码如下：

复制代码

```
#ifdef GL_ES
precision highp float;
#endif

uniform sampler2D tMap;
uniform sampler2D tCat;
varying vec2 vUv;

void main() {
    vec4 color = texture2D(tMap, vUv);
    vec2 st = vUv * 3.0 - vec2(1.2, 0.5);
    vec4 cat = texture2D(tCat, st);

    gl_FragColor.rgb = cat.rgb;
    if(cat.r < 0.5 && cat.g > 0.6) {
      gl_FragColor.rgb = color.rgb;
    }
    gl_FragColor.a = color.a;
}
```

如上面的代码所示，我们可以先通过 tCat 纹理获取绿幕图片。如果 RGB 通道中的 G 通道超过阈值，且 R 通道低于阈值，我们就可以接着把猫的图像从纹理中定位出来。然后经过缩放和平移变换等操作，我们就能把它放置到画面中适当的位置。

要点总结

今天，我们讨论了边缘模糊和纹理叠加这两种滤镜，并且重点学习了用 Shader 加载纹理和实现滤镜的方法。

首先，我们知道了什么是边缘模糊，边缘模糊很容易实现，只要我们在遍历像素点的时候，同时计算当前像素点到图片中心点的距离，然后根据距离设置透明度，就可以实现边缘模糊的效果。

然后， 我们重点讲了 Shader 中的纹理叠加滤镜。

要实现这个滤镜，我们要先加载纹理，获取纹理的颜色。用 Shader 加载纹理的过程比较复杂，但我们可以使用一些封装好的库，如 gl-renderer 来简化纹理的加载。那在获取纹理的颜色的时候，我们可以通过 texture2D 函数读取纹理单元对应的 uv 坐标处的像素颜色。

加载了纹理之后呢，我们就可以通过纹理结合滤镜函数来处理像素，这就是纹理滤镜的应用场景了。通过纹理滤镜，我们不仅可以实现灰度化图片，还可以图片的粒子化渐显等等更加复杂的效果

除此之外，我们还可以使用 shader 加载多个纹理图片，把它们的颜色按照不同的方式进行叠加，从而实现图像合成。图像合成虽然在可视化中使用得比较少，但它非常适合用来实现一些特殊的视觉效果。

小试牛刀

1. 你可以完善一下片元着色器中的颜色滤镜函数，实现灰度效果以外的效果吗？
2. 上节课，我们用 Canvas2D 实现了平滑效果滤镜，其实我们也可以用 Fragment Shader 结合纹理的形式把它实现出来，你能做到吗？
3. 如果我们想让一个图片的某个局部呈现“马赛克”效果，该用什么滤镜？你能把它实现出来吗？

欢迎在留言区和我讨论，分享你的答案和思考，也欢迎你把这节课分享给你的朋友，我们下节课见！

源码

课程示例代码

推荐阅读

Texture 的参数设置参考文档

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 12 | 如何使用滤镜函数实现美颜效果？

下一篇 14 | 如何使用片元着色器进行几何造型？

精选留言 (1)

写留言

木瓜777
2020-07-22

shader中有很多数字，为什么是这些数字？

展开

作者回复: 如果你说的是随机数函数里的数字，那个数字就是随便写的一个大数，因为伪随机用正弦函数小数点后几位来模拟随机

1